

Lab 2 Report – Software Interrupt

Kunjian Song

1 SWI Handler Design

The SWI handler consists of the four blocks:

1. Preserve the user mode registers, i.e. copy the register values in **{r0-r12}** and link register **lr** to stack and update the stack pointer **sp**.
2. Compare the SWI address with 0xff. If not equal, the program shall not serve it. Simply return to the caller.
3. If equal, the program shall carry on and print the characters.
4. Restore the user mode registers and return

In Step 3, the code to print out the characters were originally from the *text_out.s* file.

1.1 Code Changes and Rationale Behind Them

In *swi.s* file, the changes were as followings:

```
9      Restore_If_Not_Exec ;soubroutine to restore the user mode if not 0xff
10      LDMFD sp!, {r0-r12,pc} ; restore user mode reg
11      MOV pc, lr ; return
12
13      My_SWI_Handler ;put your swi handler code here
14
15          STMFD sp!, {r0-r12,lr} ; preserve user mode reg
16
17          MOV     r0, lr           ; load the link reg that point to the SWI instruction + 4 byte
18          SUBS    r0, r0, #4       ; subtract 4 to get last instruction
19          LDR     r1, [r0]         ; load the instruction
20          AND     r1, r1, #0xff    ; extract the last byte with an mask
21          CMP     r1, #0xff        ; compare with value 0xff
22          BNE     Restore_If_Not_Exec ; branch when unequal
23
24          MOV     r0, #0x3         ;select Angel SYS_WRITEC function
25      NxtTxt  LDRB    r1, [r14], #1 ;get next character
26          CMP     r1, #0           ;test for end mark
27          SUBNE   r1, r14, #1      ;setup r1 for call to SWI
28          SWINE   SWI_ANGEL        ;if not end, print..
29          BNE     NxtTxt           ; ..and loop
30          ADD     r14, r14, #3      ;pass next word boundary
31          BIC     r14, r14, #3      ;round back to boundary
32
33          LDMFD sp!, {r0-r12,pc} ; restore user mode reg
34
35          MOVS    pc, lr           ;return
```

Figure 1: Changes in SWI handler

Line 15 STMFD command was used to push the contents in **{r0-r12, lr}** onto a full descending stack and updated the stack pointer (**sp**) using the exclamation mark. Here curly braces were used to denote a list of registers.

Line 17 – 22 will be discussed in Section 1.2 in more details.

Line 24 – 31 were from *text_out.s* file. **Register r14 (lr)** was used to pass the character to the function because the address of the instruction below the “SWI 0xff” instruction is stored in r14 as shown in the Figure 2, i.e. pointing to the address PC + 4 bytes.

23	SWI	0xff
24	=	<u>"Test string1", &0a, &0d, 0</u>
25	SWI	0xff
26	=	"Alternative test", &0a, &0d, 0
27	SWI	0xfa
28	=	"unprintable string", &0a, &0d, 0
29	Exit	;finish

Figure 2: r14 stores the address of this instruction in the memory

This is why r14 is in square bracket, which means **register indirect addressing mode** is used in Line 25 in Figure 1, loading the characters (byte after byte) as indicated in the offset “#1”.

Line 33 restores the registers for user mode by popping them out from the full descending stack and update the stack pointer (sp) after execution.

Line 35 returns to the caller. MOVS instruction was used here to set the conditional flag.

1.2 Comparing the SWI address with 0xff

We know that the link register (lr) stores the return address of the next instruction in the caller routine. By disassembling *swi_test.s* file, it shows that we need to subtract 4 bytes to get the instruction we want (underlined in blue colour).

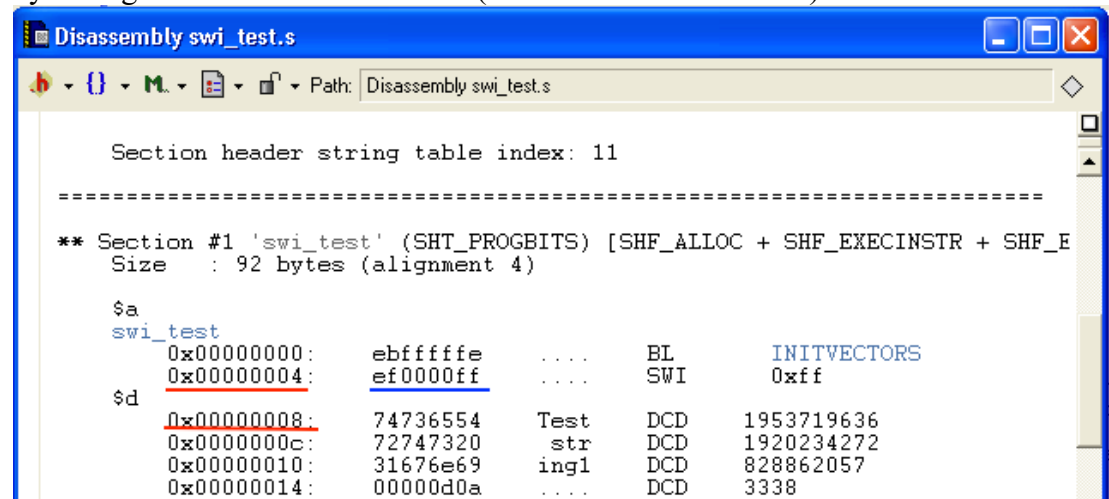


Figure 3: Disassemble of swi_test.s

ARM instruction is 32-bit. Hence the PC counter is increased by 4 byte every time. This is why we need to subtract 4 bytes in Line 18 in Figure 1. This logic can also be confirmed in the memory view in AXD debugger using Processor Views -> Memory.

ARM7TDMI - Memory		Start address: 0x8000															
Tab1 - Hex - No prefix				Tab2 - Hex - No prefix				Tab3 - Hex - No prefix				Tab4 - Hex - No prefix					
Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	ASCII
0x00008000	28	00	00	EB	FF	00	00	EF	54	65	73	74	20	73	74	72	(.....Test str
0x00008010	69	6E	67	31	0A	0D	00	00	FF	00	00	EF	41	6C	74	65	ingl.....Alte
0x00008020	72	6E	61	74	69	76	65	20	74	65	73	74	0A	0D	00	00	rnative test....
0x00008030	FA	00	00	EF	75	6E	70	72	69	6E	74	61	62	6C	65	20unprintable
0x00008040	73	74	72	69	6E	67	0A	0D	00	00	00	00	18	00	A0	E3	string.....
0x00008050	00	10	9F	E5	56	34	12	EF	26	00	02	00	FF	9F	BD	E8V4..&.....
0x00008060	0E	F0	A0	E1	FF	5F	2D	E9	0E	00	A0	E1	04	00	50	E2_......P.
0x00008070	00	10	90	E5	FF	10	01	E2	FF	00	51	E3	F6	FF	FF	1AQ.....
0x00008080	03	00	A0	E3	01	10	DE	E4	00	00	51	E3	01	10	4E	12Q...N.
0x00008090	56	34	12	1F	FA	FF	DE	1A	03	E0	8E	E2	03	E0	CE	E3	V4.....
0x000080A0	FF	9F	BD	E8	0E	F0	B0	E1	00	80	A0	E3	10	90	8F	E2
0x000080B0	FF	00	B9	E8	FF	00	A8	E8	FF	00	B9	E8	FF	00	A8	E8
0x000080C0	0E	F0	A0	E1	18	F0	9F	E5	18	F0	9F	E5	18	F0	9F	E5

Figure 4: The part underline is the instruction "SWI 0xff" stored in memory starting at 0x8000

In Line 19 Figure 1, register indirect addressing mode is used again to load the memory content pointed by the address stored in r0. In Line 20, AND operation was used to extract the 4th byte in that instruction, followed CMP instruction. Based on previous Cortex-M assembly experience, the CMP instruction actually uses subtraction to compare two values and set the flag in APSR (Application Program Status Register). BNE instruction will check APSR and branch to the Resotre_If_Not_Exec subroutine (Line 9 in Figure 1).

1.3 Changes in vectors.s File

Changes are in Line 4 and Line 32 as shown in Figure 5 below.

```

1      AREA    vectors, CODE, READWRITE
2      EXPORT  INITVECTORS
3      EXTERN My_SWI_Handler
4
5      ;install the exception handlers at reset
6      ;by copying a precompiled block from RAM to adress 0
7
8  INITVECTORS
9      MOV     r8, #0
10     ADR     r9, Vector_Init_Block
11     LDMIA   r9!, {r0-r7}           ;Copy the vectors (8 words)
12     STMIA   r8!, {r0-r7}
13     LDMIA   r9!, {r0-r7}           ;Copy the DCDed addresses
14     STMIA   r8!, {r0-r7}           ;(8 words again)
15     MOV     pc, r14
16
17
18
19     Vector_Init_Block
20     LDR     PC, Reset_Addr
21     LDR     PC, Undefined_Addr
22     LDR     PC, SWI_Addr
23     LDR     PC, Prefetch_Addr
24     LDR     PC, Abort_Addr
25     NOP                                ;Reserved vector
26     LDR     PC, IRQ_Addr
27     LDR     PC, FIQ_Addr
28
29
30     Reset_Addr    DCD    Default_Start_Boot    ;Useless when using the debugger
31     Undefined_Addr DCD    Default_Undefined_Handler
32     SWI_Addr      DCD    My_SWI_Handler

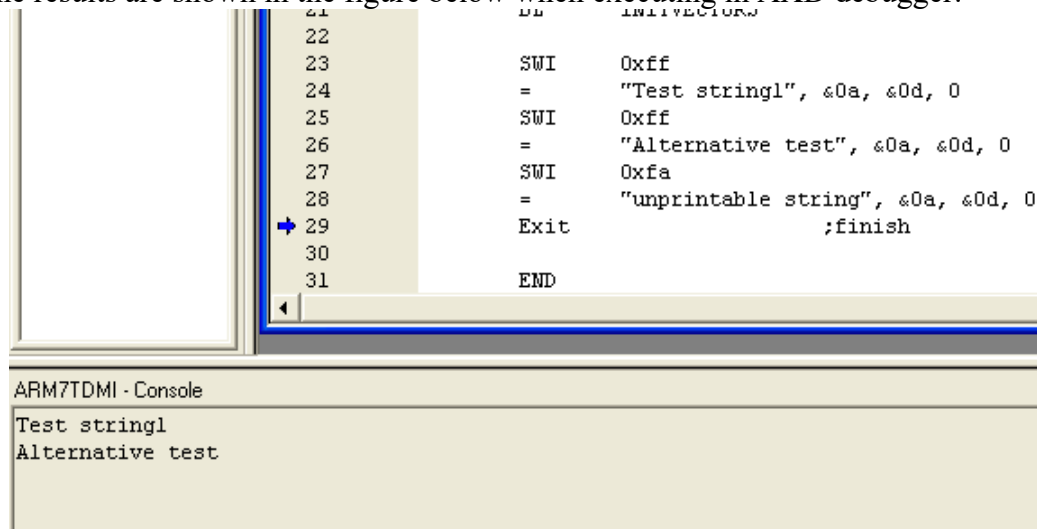
```

Figure 5: Changes in vectors.s file

In Line 32, the address of My_SWI_Handler is DCDed in SWI_Addr. DCD directive defines the label, SWI_Addr, as the SWI handler. When program has branched in to INITVECTORS subroutine which further calls Vector_Init_Block subroutine which loads the starting address of My_SWI_Handler to the PC in Line 22 in Figure 5. (Note that it also loads some other handlers but they are not defined. Hence nothing will be executed.)

2 Program Results

The results are shown in the figure below when executing in AXD debugger:



The screenshot displays the AXD debugger interface. The top pane shows assembly code with line numbers 21 through 31. Line 22 is highlighted with a blue arrow. The code includes SWI instructions with various arguments and an Exit instruction. The bottom pane, titled 'ARM7TDMI - Console', shows the output of the program: 'Test string1' and 'Alternative test'.

```
21      DD      INITVECTORS
22
23      SWI      0xff
24      =      "Test string1", &0a, &0d, 0
25      SWI      0xff
26      =      "Alternative test", &0a, &0d, 0
27      SWI      0xfa
28      =      "unprintable string", &0a, &0d, 0
29      Exit
30      ;finish
31      END
```

ARM7TDMI - Console
Test string1
Alternative test

Figure 6: Results

As shown in Figure 6, the “SWI 0xfa” was not “handled” due to the address not being 0xff.

For more details, please refer to the code zipped and submitted along with this report.